Comp. & Maths. with Appls. Vol. 12A, No. 9, pp. 987–990, 1986
Printed in Great Britain.
0886-9553/86 $3.00 + 0.00

PARALLEL ALGORITHMS FOR SOLVING LINEAR EQUATIONS USING GIVENS TRANSFORMATIONS

M. HEAD-GORDON[1] and P. PIELA[2]
Departments of [1]Chemistry and [2]Chemical Engineering, Carnegie-Mellon University, Pittsburgh, PA 15213, U.S.A.

(*Received January 1986*)

Communicated by G. J. Fix

Abstract—The use of the Givens method to solve linear equations on a parallel computer is reviewed, and a new algorithm which requires fewer time steps in the infinite processor case is presented.

1. INTRODUCTORY REVIEW

This paper is a discussion of the use of Givens transformations to solve systems of linear equations on massively parallel computers. After a brief review of Givens transformations in this section, a new parallel algorithm is presented in Section 2, and compared with the existing algorithm of Sameh and Kuck [1].

A Givens transformation is a plane rotation on two rows of a matrix which introduces a zero into one row. Thus, $\mathbf{G}(i, j, k)$ rotates rows i and j to zero the element at x_{jk}:

$$\mathbf{G}(i,j,k)\mathbf{x} = \begin{bmatrix} 1 & & & & \\ & \ddots & & & \\ & & C & S & \\ & & & \ddots & \\ & & -S & C & \\ & & & & \ddots \\ & & & & & 1 \end{bmatrix} \begin{bmatrix} 0 \dots x_{ik} & x_{ik+1} \dots \\ 0 \dots x_{jk} & x_{jk+1} \dots \end{bmatrix} = \begin{bmatrix} 0 \dots x'_{ik} & x'_{ik+1} \dots \\ 0 \dots 0 & x'_{jk+1} \dots \end{bmatrix}, \quad (1)$$

where

$$C\,(=\cos\theta) = x_{ik}/\sqrt{x_{ik}^2 + x_{jk}^2} \quad (2)$$

and

$$S\,(=\sin\theta) = x_{jk}/\sqrt{x_{ik}^2 + x_{jk}^2}. \quad (3)$$

Columns which initially have zeros in both rows i and j retain them after transformation. Givens rotations are numerically very stable [2] and are a good way of selectively introducing zeros into a matrix.

The standard Givens transformation (1)–(3) is inefficient relative to alternatives like Householder reflections. However, the fast Givens transformation introduced by Gentleman [3], which does not use pure rotations and requires only half as many multiplications, is fairly competitive [2]. As it still involves pairwise transformation of rows, the following discussions of Givens based algorithms do not need to assume which elementary transformation is used.

A serial algorithm to solve an augmented matrix using a series of Givens transformations might typically proceed as follows:

(a) row 1 is used to successively rotate to zero all elements in column 1 below the diagonal;
(b) row 2 is used to successively zero all elements below the diagonal in column 2;
(c) this process is continued until the matrix is triangularized and can then be solved by back-substitution.

This algorithm is intrinsically serial because new rotations usually require the modified row generated by the previous rotation, so that only one can be done at a time. On serial computers, the Givens method is generally only competitive with conventional equation solvers when the coefficient matrix is sparse.

2. THE GIVENS METHOD IN THE PARALLEL SETTING

The Givens method is of interest in parallel computing because pivoting which can dominate parallel Gauss–Jordan and Gaussian elimination algorithms [4] is not required. A parallel implementation of the Givens method must take advantage of the fact that a single rotation (or fast Givens transformation) affects only two rows. Hence, it is possible to simultaneously generate additional zeros using other distinct pairs of rows. If the matrix has dimension N, then a maximum of $\lfloor N/2 \rfloor$ rotations can be performed at once.

We shall discuss parallel algorithms assuming an infinite number of processors, by considering the number of "transformation time steps" T_t involved, which is the number of time steps assuming an elementary rotation takes a single step. For example, $T_t = \frac{1}{2}N(N-1)$ in the serial Givens method; this being the number of zeros that must be generated one at a time. Using T_t means that we can concentrate on the conceptual structure of the algorithms rather than details of program implementation.

Firstly, we shall consider the algorithm of Sameh and Kuck (S–K) [1], reported in Ref. [4]. The augmented matrix is triangularized by starting from the lower left-hand corner and generating a step structure of zeros which advances across the matrix. Adjacent rows forming a step are used in pairwise fashion to generate new zeros, and the elegance of the algorithm is that the step structure is retained at each transformation time step. As an example, Fig. 1 illustrates the sequence in which zeros are generated in the $N = 8$ case. Several steps are explicitly shown to illustrate the pairing of rows and the numbers in the lower matrix show at which transformation time step a zero is generated at each position. $T_t = 13$, and the degree of parallelism (as measured by the number of simultaneous rotations) rises from 1 to its maximum of $\lfloor N/2 \rfloor$ at the halfway point, and then falls symmetrically. This is characteristic of the S–K algorithm.

Appendix A contains an implementation of the S–K Givens algorithm from Ref. [4]; where rotate$(i, j, k) = \mathbf{G}(i, j, k)\mathbf{x}$. Parallelism occurs in the first rotate statement where each value of p can be processed concurrently, and similarly in the second rotate statement. The number of processors required, assuming fast Givens transformations are used, is $P =$ (maximum parallelism) × (processors needed for a rotation)

$$= \frac{N}{2} \times 2(N+1) = N(N+1). \tag{4}$$

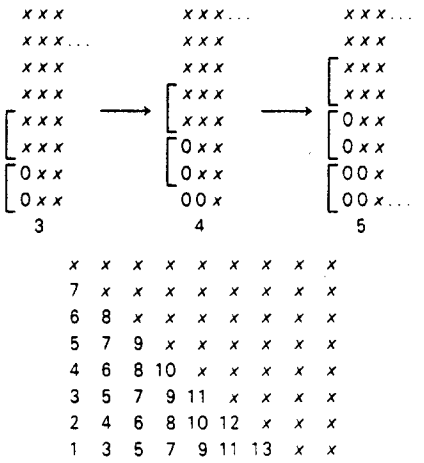

Fig. 1. S–K algorithm for $N = 8$ explicitly showing Steps 3–5.

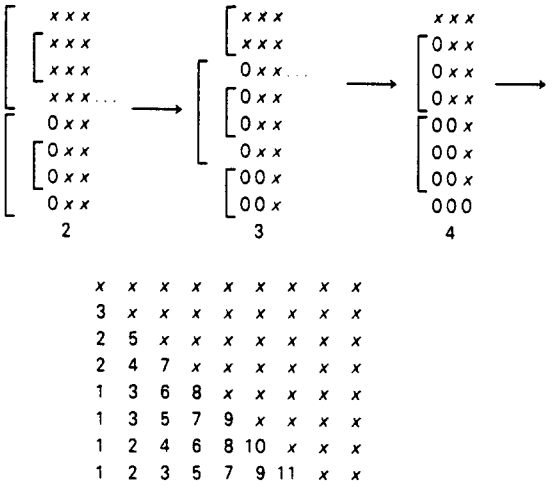

Fig. 2. H–P algorithm for $N = 8$, explicitly showing Steps 2–4.

The number of transformation time steps required in the massively parallel case is

$$T_t(N) = 2N - 3, \tag{5}$$

which follows from putting a zero in a new column on every second time step, including the first and last, and having $N - 1$ columns to do. Since, as is evident in Fig. 1, the region of zeros grows twice as fast vertically as horizontally, then when a zero is generated in the $(N-1)$st column at the $(2N-3)$rd transformation time step the triangularization is complete. Parallel methods for back-substitution are discussed in Ref. [4].

We now present an alternative parallel algorithm developed by the authors and denoted as H–P. At each transformation time step as many new zeros as possible are generated in the spirit of a fan-in algorithm. For example, Fig. 2 is the $N = 8$ case, for comparison with Fig. 1. The step structure of the S–K algorithm ensures correct pairing of rows to retain previously created zeros, whereas in the H–P algorithm the pairing is different, as shown in Steps 2–4 of Fig. 2. $T_t = 11$, which is 2 less than the S–K algorithm. The H–P algorithm attains maximum parallelism immediately unlike the S–K Givens, and thereafter the degree of parallelism decreases. From about Step 6 onwards, the step structure characteristic of the S–K Givens emerges and the final operations are identical for both algorithms.

The number of transformation time steps involved in the H–P algorithm can be deduced with reference to Fig. 2. The initial fan-in of Step 3 allows us to begin working on Column 2 in Step 2 and Column 3 in Step 3. Thereafter a zero can be placed in a new column on every second time step, reflecting the need to first obtain two zeros in the previous column which takes two steps. IF $n = \lfloor \log_2 N \rfloor$ then the combination of these processes gives

$$T_t(N) = 2N - (2 + n), \tag{6}$$

which can be viewed as the $2N - 3$ steps of the S–K algorithms with an $(n - 1)$ step saving due to the initial fan-in. Appendix B is an implementation of the H–P algorithm, where all operations in the loop over i can be performed in parallel, and the number of floating point operations and processors required is the same as for the S–K algorithm, although slightly more overhead is involved.

By way of conclusion we note that the H–P algorithm introduced here offers a saving of $n - 1$ transformation time steps (or 2 if $N = 2^3$) over the S–K algorithm in a massively parallel setting. In the p processor case, the download algorithm [5] predicts that if w operations are involved then the p processor time T^p is related to the time steps in the infinite processor case T^∞ by

$$T^p \leqslant T^\infty + (w - T^\infty)/p. \tag{7}$$

These are now numbers of true time steps, and as $w(\text{H–P}) = w(\text{S–K}) \gg T^{\infty}$, the percentage difference in download times between these algorithms becomes negligible when p is small. Additionally, when p is small, pivoting in Gauss–Jordan and Gaussian elimination no longer dominates the computation, and these methods are then viable.

REFERENCES

1. A. H. Sameh and D. J. Kuck, Linear system solvers for parallel computers. Dept of Computer Science, Univ. of Illinois, Urbana, Ill. (1975).
2. G. H. Golub and C. F. Van Loan, *Matrix Computations*, 1st edn. John Hopkins Univ. Press, Baltimore, Md (1983).
3. W. M. Gentleman, Least squares computations by Givens transformations without square roots. *J. Inst. Math. Applic.* **12**, 329–336 (1973).
4. D. Heller, A survey of parallel algorithms in numerical linear algebra. *SIAM Rev.* **20**, 740–777 (1978).
5. R. P. Brent, The parallel evaluation of general arithmetic expressions. *J. Ass. comput. Mach.* **21**, 201–206 (1974).

APPENDIX A

Implementation of the S–K Algorithm

Rotate(i, j, k) = G(i, j, k)x, as defined in equation (1)

```
for: k = 1 to N − 1
    for p = 0 to min{k − 1, N − k − 1}
      rotate(N − 2p − 1, N − 2p, k − p)
    end
    do p = 0 to min{k − 1, N − k − 2}
      rotate(N − 2p − 2, N − 2p − 1, k − p)
    end
end
```

APPENDIX B

An Implementation of the H–P Algorithm

```
n = ⌊log2N⌋
upper-row (1) = 1
back-col = 1
lower-row (col) = N, col = 1,N
for i = 1 to 2N − (2 + n)
    front-col = min{i,n + (i − n)/2}
    for col = back-col to front-col
      for row = 0, [lower-row(col) − upper-row(col) + 1]/2 − 1
        rotate[upper-row(col) + row, lower-row(col) − row, col]
      end
    end
    for col = back-col to front-col
      lower-row(col) = lower-row(col) − [1 + lower-row(col) − upper-row(col)]/2
    end
    for col = back-col to front-col
      upper-row (col + 1) = max[lower-row(col) + 1, col + 1]
    end
    back-col = back-col + max{0, upper-row(back-col) − lower-row(back-col) + 1}
end
```